Galios Connection and Abstract Interpretation

Aravind Machiry
amachiry@purdue.edu

1 Credits

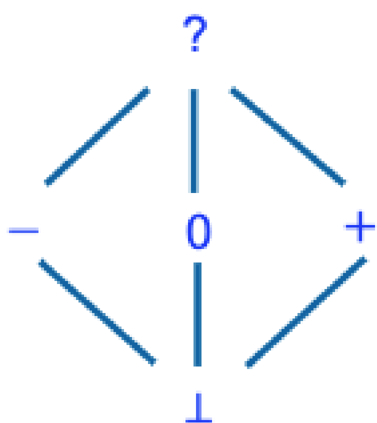
Copied from Elizabeth Dinella (UPenn)

2 Abstract Interpretation

Any non-trivial semantic property that we may wish to check about programs is undecidable. Examples of such properties include whether an assertion may fail on some input, whether the program will terminate on all inputs, and many others. Thus, it is clear that we must work with approximations of programs, called `abstractions`.

Designing a suitable program abstraction is a delicate balance of tradeoffs. By abstracting a program, we lose some concrete information. It is important to ensure that our approximation is sound, meaning that all claims made in the abstract are always valid in the concrete. Abstract Interpretation [1] is a theory of approximation that can be applied to systematically construct sound analysis algorithms. It guarantees sound but potentially incomplete abstractions.

2.1 Abstract Domain: The Sign Domain



The sign abstract domain consists of five abstract values as follows:

- Zero (0), representing the integer value 0;
- Minus (-), representing any negative integer value;
- Plus (+), representing any positive integer value;

- Top (?), representing unknown; and
- Bottom ($\perp$), representing uninitialized by the analysis.

In contrast, the concrete domain contains all possible integer values. Both the abstract and concrete domain are lattices. A lattice is a mathematical term for a partially ordered set (poset). This means there must be a concept of order, however, not every pair of elements need be comparable. Additionally, for a set to be a lattice, every pair of elements must have a "greatest lower bound" and a "least upper bound." Formally, an element a is an "upper bound" for subset A if for all b in A, $b \leq a$. Furthermore, an element a is a "least upper bound" for subset A if a is an upper bound and for all upper bounds u in A, $a \leq u$. The definitions of lower bound and greatest lower bound follow similarly.

The theory of abstract interpretation rests on the assumption that the abstract and concrete domains are indeed lattices. Our abstract domain has a least upper bound and a greatest lower bound for each pair of elements and is indeed a lattice.

In addition, we must define an abstraction function (α) that maps elements of the concrete domain to the abstract domain and a concretization function (γ) that maps elements of the abstract domain back to concrete integer values.

The abstraction function for this domain is quite simple:

$$\alpha(S) = \begin{cases} 0 & \text{if all elements of S are 0} \\ + & \text{if all elements of S are positive} \\ - & \text{if all elements of S are negative} \\ ? & \text{otherwise} \end{cases}$$

Likewise, the concretization function is defined as follows:

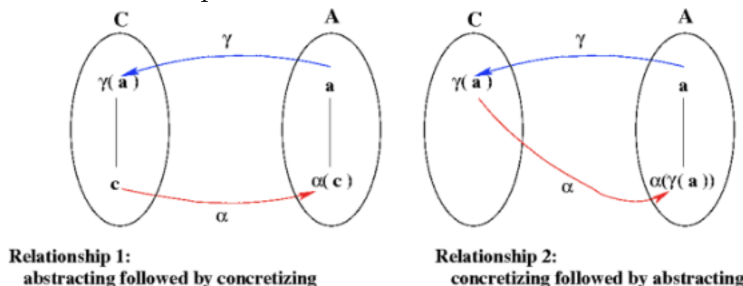
$$\gamma(S) = \begin{cases} \{0\} & \text{if } S = 0 \\ \{\text{pos int}\} & \text{if } S = + \\ \{\text{neg int}\} & \text{if } S = - \\ \{0 \text{ pos neg}\} & \text{if } S = ? \end{cases}$$

In order for a sound approximation, these functions must form a galois connection. A galois connection is a pair of functions between two partially ordered sets such that the following properties hold:

For all a in the partially ordered abstract set A, and all c in the partially ordered concrete set C,

- $\alpha(\text{c}) \leq a \iff c \leq \gamma(a)$
- $\alpha(\gamma(a)) \leq a$

These relationships are shown below:



In short, in order to define an abstract interpretation, we must

1. Define the abstraction domain, an abstraction function α, and concretization function γ such that α and γ form a galois connection. We have already shown this for our abstract domain!

2. Define a monotonic abstract semantics - this means that if the approximation $\alpha(c)$ is an overapproximation of the concrete element c, then after applying transfer functions, the overapproximation should still hold. This will become more clear as we define an abstract semantics for our sign domain.

If these properties are satisfied, the program analysis is guaranteed to be sound.

Now, returning to our example, recall our simple sign domain. Although it has only a few states, it satisfies the requirements for a sound abstract interpretation and can do the job for some straightforward use cases. For example, consider the following program: The text in blue (x:) represent the abstract value of x at the current program point. The comments represent specific points in the program.

```
  x:⊥
if (x == 0) {       // p1
  x: 0
 x++;             // p2
  x: +
} else if (x > 0) { // p3
  x: +
 x = x * 20;      // p4
  x: +
} else if (x < 0) { // p5
  x: -
 x = x * (-10);   // p6
  x: +
}
  x: +
assert(x > 0);
```

At the start of the program, the analysis initializes the abstract value of x to $\bot$. Next, we interpret the conditional at program point $p1$ to determine that the abstract value of x should be 0. At $p2$ we reach the addition operator. How should we apply addition to an abstract value? In general, we must define the semantics of each abstract operator. The abstract addition operator can be defined as expected: adding two negative values results in a negative number, but adding a negative value to a positive value results in an unknown sign.

The rest of the semantics for abstract addition are shown in the following table:

ADD	-	0	+	?
-	-	-	?	?
0	-	0	+	?
+	?	+	+	?
?	?	?	?	?

So, at program point $p2$ the abstract value of x becomes $+$. Similarly, at program point p3 we interpret the conditional and determine that the abstract value of x should be $+$. Now,

we reach program point $p4$ that involves multiplication. Likewise, we must define an abstract multiply operator:

MULT	-	0	+	?
-	+	0	-	?
0	0	0	0	0
+	-	0	+	?
?	?	0	?	?

So, at program point $p4$ we set the abstract value of x to be $+$, the result of multiplying two positive values. The last conditional follows similarly. Once we reach the assertion, we must merge the abstract values of x from each path. As hinted by our abstraction and concretization functions, the merge operator in this domain is a conjunction. This means that for the abstract value to be $+$, the abstract values of all incoming paths must also be $+$. If there is any disagreement, we default to ?.

It is clear that a merge may cause us to lose some information. For example, consider the following program:

```
  x:⊥
if (x == 0) {         // p1
  x: 0
 x++;               // p2
  x: +
} else if (x > 0) { // p3
  x: +
 x = x * (-10);   // p4
  x: -
}
x: ?
assert(x != 0);
```

Like the previous example, the abstract value for x starts unitialized and is set by interpreting conditionals and applying the abstract addition and multiplication operators. However, in this example, when the two program paths merge we have differing abstract values for x. We know that in one program path, the value of x is positive, and in the other program path, the value of x is negative. Thus, it is clear that the value is nonzero. Unfortunately, our abstract domain does not have a state to represent that the value could be $+$ or $-$ but not 0. When we merge, the abstract value for x is ? even though it is clear that the value of x will never be 0 when we reach the assertion. This provides some motivation for a more precise abstract domain.

References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.